DPADL: An action language for data processing domains

Keith Golden
NASA Ames Research Center
MS 269-2
Moffett Field, CA 94035
kgolden@email.arc.nasa.gov

Abstract

This paper presents DPADL (Data Processing Action Description Language), a language for describing planning domains that involve data processing. DPADL is a declarative, object-oriented language that supports constraints and embedded Java code, object creation and copying, explicit inputs and outputs for actions, and metadata descriptions of existing and desired data. DPADL is supported by the IMAGEbot system, which we are using to provide automation for an ecosystem forecasting system called TOPS.

1 Introduction

NASA is a data-collection agency. Terabytes of data are gathered each day by NASA telescopes, satellites and other spacecraft, and processing these data is challenging. Current approaches to data processing were devised when data volumes were small and it was possible for scientists to "look at" all the data. With the large data volumes now available, increasing levels of automation are needed if scientists are to avoid drowning in data. This is especially true in the Earth Science Enterprise, where high bandwidth communication makes it practical to return terabytes per day, and where the wide variety of instruments and the even wider variety of uses of the data make scripted approaches to automation too labor intensive.

We are working with Earth scientists to help address this challenge in the context of an ecosystem forecasting system called the Terrestrial Observation and Prediction System, or TOPS (Nemani *et al.* 2002) (http://www.forestry.umt.edu/ntsg/Projects/TOPS/). Our approach is to cast the problem of automating data-processing operations as a planning problem. We have developed a planner-based softbot (**soft**ware ro**bot**), called IMAGEbot, to generate and execute dataflow programs (plans) in response to data requests. The data processing operations supported by IMAGEbot include image processing, text processing, managing file archives and running scientific models. Aspects of the planner are described in (Golden & Frank 2002). In this paper, we describe the IMAGEbot action language, which we call DPADL, for Data Processing Action Description Language. DPADL is a successor to the ADLIM language, described in (Golden 2000). A key requirement of the language is to provide precise causal models of data-processing actions, which can be used either for plan generation, or to derive metadata descriptions of the outputs of a given plan.

Given the richness and complexity of software systems and data-processing programs, the language used to describe them must be expressive. Following are some features it should support.

- First-class objects: Most things in the world and in software environments can be viewed as objects with certain attributes and relations to other objects. For example, a person may be described in terms of name, address, workplace, etc., while a file has a name, host location, owner, etc. Even more importantly, data files typically have a complex, hierarchical structure, which can be described in terms of object composition. The language should support structures or objects as first-class citizens, to reflect the underlying structure of the software environments it is meant to describe.
- Object creation and copying: Many programs create new objects, such as files, sometimes by copying or modifying other objects. The language should have a simple way of describing such operations.
- Operations on sets of objects: Many programs act on all members of some set. For example, "`lpr *`" prints all files in the current directory, and an image processing operation may affect all pixels in an image or a specified region of an image. The language should support universal quantification to describe such operations.
- Integration with a run-time environment: It is not sufficient to generate plans; it is necessary to execute them, so there must be a way to describe how to execute the operations provided by the environment and obtain information from the environment. The language should allow the specification of "hooks" into the runtime environment, both to obtain information and to initiate operations.
- Constraints: Determining the appropriate parameters for an action can be challenging. Parameter values can depend on other actions or objects in the plan. The language should provide the ability to specify such constraints where they are needed.

We have developed a language for describing objects and actions in a data-processing domain. The language provides tight integration with the Java run-time environment. In particular, Java code can be embedded in the action descrip-

tions, planner variables can be bound to Java objects as well as primitive types, and constraints can be enforced by invoking methods on those objects.

In the remainder of the paper, we describe the language in detail. The elements of a domain description include specifications of types (Section 2), functions and relations (Section 3), goals (Section 5), actions (Section 6) and states (Section 7). Definitions of types and functions can include constraints (Section 4).

At the end of each section, we present a BNF grammar covering the language elements described in that section. The boxes containing these descriptions may be skipped by readers unfamiliar with BNF notation or uninterested in the formal specification.

For example, we have stated so far that domain descriptions include types, function, goals, actions and states, and can include embedded Java code. The top-level production rule for a domain description is:

DOMAIN ::=	(TYPE | FUNCTION | ACTION | GOAL | STATE | <INLINE_CODE>)+ <EOF>

where symbols in SMALL CAPS are non-terminals, symbols in <ANGLE_BRACKETS> are terminals, and keywords are underlined.

2 Objects and primitive types

The language provides several predefined types, and supports the declaration of new types derived from the predefined types. The keyword for introducing a new type declaration is **type**. Here and elsewhere in the paper, DPADL text is rendered in `typewriter` font, and keywords are **bold**. We use ellipses (...) to indicate that text has been omitted for the sake of brevity. For example,

```
static type Filename isa string
```

introduces a new type, `Filename`, which is a subtype of `string`, a predefined type. The predefined types are int, unsigned, real, string, object and boolean. The keyword **static** means that no instance of `Filename`, once created, can ever be changed. A type that is not static is **fluent**.

Subtypes of `object` may be used to represent Java objects. For example,

```
static type Tile isa object
            mapsto tops.modis.Tile
```

means that the type `Tile` corresponds to the Java class `tops.modis.Tile`. As we discuss in Sections 4 and 6.4, the agent can manipulate Java objects in the course of constraint reasoning or action execution by executing inlined Java code.

Alternatively, we can define a type by listing all possible instances of the type. This is similar to enumerated types in C/C++, but without the restriction to integral values.

```
static type ImageFormat =
{"JPG", "GIF", "TIFF", "PNG", "XCF", ...};
```

As in C/C++, enumerated values can have symbolic names attached to them.

```
static type ProjectionType =
{LAZEA=11, GOODE_HOMOL=24, ROBINSON=21, ...};
```

Like classes in C++ and Java, types can have attributes. For example, file attributes include pathname and owner:

```
type File isa object {
  static key Path pathname;
  User owner;
  ...
}
```

The keyword **key** is used to indicate that `pathname` is a unique identifier for a file, so two files that have the same pathname must in fact be the same file. Identifying the pathname as the key precludes the possibility of moving or renaming a file, or copying one file on top of another, since those operations change the pathname without changing the identity of the file, or vice versa. A more robust choice for a key is the file's "inode number," a serial number assigned to the file that stays the same even if the name or directory changes. Even that is imperfect, however, since two files stored on different computers could have the same inode number. We can solve this problem by making the host machine an additional key:

```
type File isa object {
  static key int inodeNumber;
  static key string hostName;
  Path pathname;
  User owner;
  ...
}
```

When an object has multiple keys, it is their combination that uniquely identifies the object, so by the above definition, two files are identical if they have the same host machine *and* inode number.

In addition to the subtype relation, designated using **isa**, we can specify that one type **implements** another, meaning it inherits all the attributes of the other type but is not an instance of that type. This is useful in cases where two objects share the same structure but cannot be used interchangeably. For example, a file archive, such as a tar file, contains records which reflect all the properties and contents of individual files, but are not themselves files. We say that `TarFile.Record` **implements** `File`. This is especially useful when used in conjunction with **copyof** (Section 6.3), since a record in a tar file can be a copy of a file, or vice versa.

When referring to an attribute of an object, we use a Java-like syntax. For example, `f.filename` refers to the `filename` attribute of the object represented by the variable `f`. Attributes can take arguments. For example, `pic.pixelValue(x,y)` refers to the value of the pixel at the x,y coordinates of the image `pic`. Although the syntax resembles that of Java method calls, `pixelValue(x,y)` is simply a parameterized attribute, and can be used in exactly the same contexts. For example,

```
pic2.pixelValue(10,10) := pic1.pixelValue(0,0);
```

describes an effect that sets the value of the pixel at coordinates 10,10 in the image `pic2` to be equal to the value of the pixel 0,0 in the image `pic1`.

Object attributes may be thought of as a functions or relations that are just written in a funny way. For example, `v = pic1.pixelValue(0,0)` could just as easily have been written as a function, `v = pixelValue(pic1, 0, 0)`, or even a relation, `pixelValue(v, pic1, 0, 0)`.

However, the object notation is not just syntactic sugar, but provides valuable information to the planner. For example, the planner can reduce search by exploiting the fact that attributes of static objects don't change once the object is created, and Section 6.3 discusses the role attributes play when objects are copied.

```
TYPE ::= (static | fluent)? type
         ((<IDENTIFIER> = { MEMBERS } )
         | (TYPESPEC)) (TYPEBODY | ;)
MEMBERS ::= ((<IDENTIFIER> =)? LITERAL)
         (, MEMBERS)?
TYPESPEC ::= PRIMTYPE | ( <IDENTIFIER> isa
         TYPENAME
         ( implements TYPENAME )* )
         ( mapsto <CLASSNAME> )?
TYPEBODY ::= { ( MEMDEF | CTRSPEC | TYPE)* }
MEMDEF ::= ( static | fluent )? key? TYPENAME
         <IDENTIFIER>
         ( PARAMS )? ( MEMBODY | ; )
MEMBODY ::= { ( CTRSPEC )* }
PRIMTYPE ::= int | unsigned | real | string |
         object | boolean
TYPENAME ::= <IDENTIFIER> | PRIMTYPE
QUALTYPE ::= TYPENAME ( . <IDENTIFIER> )*
```

3 Functions and relations

As we said above, attributes may be viewed as functions or relations. We can also have functions and relations that do not correspond to object attributes. For example,

```
fluent real elevation(real lon, real lat);
```

declares a function that takes two real values, representing longitude and latitude, and returns a real value representing elevation. There is no special syntax for defining relations; they are just functions whose value is type boolean. Functions, like attributes, may have zero arguments, in which case the parentheses are omitted. For example,

```
fluent Date currentDate;
```

specifies that `currentDate` is a fluent function taking no arguments. A fluent with no arguments may be considered a variable, and a static function with no arguments is essentially a constant.

```
FUNCTION ::= (static | fluent) TYPENAME
         ( <IDENTIFIER>
         | <OPERATOR> ) ( ( PARAMS? ) )?
         ( ; | { ( CTRSPEC )* } )
PARAMS ::= ( PARAMDEF ( , PARAMS )? )
         | :rest PARAMDEF
PARAMDEF ::= QUALTYPE <IDENTIFIER>
```

4 Constraints

IMAGEbot uses a constraint-based planner to reason about the complex dependencies and interactions among actions and objects in the plan. Formally speaking, a *constraint* is simply a relation that holds over a set of variables, so it is straightforward to view functions, object attribute/value assignments, and even types as constraints. However, thus far, we have only shown how to *declare* functions, attributes and types, not (with the exception of enumerated types) how to *define* them. To make use of a constraint reasoning system, we need definitions, not just declarations. For example, consider the following declaration.

```
real foo(real x);
```

Given the value of `x`, we know there must be some value `y = foo(x)`, but we have provided no way to determine what that value is.

We provide two alternative ways of specifying the definition of a constraint; it may be selected from a library of predefined constraint definitions or defined in terms of arbitrary Java code embedded in the type and function declarations. The constraint reasoning system supports constraints over all primitive types as well as Java objects. It can also handle constraints involving universal quantification, as discussed in (Golden & Frank 2002).

Constraint definitions can only be given for statics. Any function defined as a constraint must be determined only by that constraint; no action may affect it.

4.1 Type constraints

Formally, a type is unary relation that is true for all instances of the type and false for all non-instances. But in the type declarations shown above, we did not define what those relations were. It is fine to say `Filename` **`isa`** `string`, but given a string, how do we know if it is a valid filename?

One possibility might be to define `Filename` as an enumerated type; that is, we list all valid filenames. The obvious problem with this is that there are, for all practical purposes, infinitely many of them. A better option is to specify a regular expression that concisely specifies all valid filenames:

```
static type Filename isa string {
   constraint Matches(true, this, "~[/]+");
}
```

means that filenames must contain at least one character, and they cannot contain the character '/'. In Unix, this is, in fact, the only practical limitation on filenames. `Matches` is a constraint from the constraint library requiring a string to a match a regular expression. The keyword **`this`** designates an instance of the type being defined, in this case a filename.

Constraints can also be defined in terms of inlined Java code, as discussed in the next section. Type constraints can also mention attributes of the type. For example, consider the following definition of Date

```
/** A date of the form MM/DD/YYYY */
static type Date isa string {
   Day day;
   Month month;
   Year year;
   constraint Concat(this, month, "/", day,
                     "/", year);
}
```

where Concat is a constraint from the constraint library specifying that the first argument is the concatenation of the remaining arguments. Using this representation, we can describe a date either as a single string or as a structure with three attributes, whichever is more convenient. Constraint reasoning can be used to convert from one representation to the other.

4.2 Attribute constraints

We can define attributes as constraints as well. One reason for doing this is to support *procedural attachment*: specifying program code that provides the definition of the attribute. For example, if we have a DPADL object that corresponds to a Java object, we must specify what methods to call on the Java object to determine the values of the attributes as declared in DPADL:

```
static type Tile isa object
            mapsto tops.modis.Tile {
  key string uniqueId {
    constraint {
      value(this) := $this.getUID()$;
      this(value) := $Tile.findTile(value)$;
    }
 ...
```

The attribute `uniqueId` is declared as a **`key`** of a (mosaic) `Tile`, meaning there is a one-to-one mapping between tiles and their unique identifiers. Given a tile, we should be able to obtain its unique identifier, and given a unique identifier, we should be able to obtain the corresponding tile. The embedded Java code provides instructions for performing these mappings. The `uniqueId` attribute of a `Tile` can be determined by calling the `getUID` method on the `Tile`, and a `Tile` object corresponding to a given `uniqueId` can be determined by calling the method `findTile`, with the `uniqueId` as an argument. The text preceding the ":=" a "signature" specifying the return value and parameters of the following Java code. The keyword **`value`** refers to the value of the attribute being defined, in this case `uniqueId`. The keyword **`this`** refers to an object of the type being defined, in this case `Tile`. Thus, **`value`**`(`**`this`**`)` means that given an object of type `Tile`, we can obtain the value of the `uniqueId` attribute by executing the following Java code (delimited by `$`). Conversely, **`this`**`(`**`value`**`)` means that given a `uniqueId`, we can find the corresponding `Tile`.

The above constraint will only be enforced if there is a singleton domain for some tile or ID variable. It is also possible to define constraints that work for non-singleton domains, by indicating that an argument or return value represents an interval (delimited by `[]`) or a finite set (delimited by `{}`). For example, one attribute of a `Tile` is that it `covers` a given longitude, latitude. Given a particular longitude and latitude, the constraint solver can invoke a method to find a single tile that covers it, but it can do even better. Given a rectangular region, represented by intervals of longitude and latitude, it can invoke a method to find a set of tiles covering that region.

```
/**
 * true if this tile covers the
 * specified lon/lat range.
 */
boolean covers(real lon, real lat) {
  constraint {

    ...
    // returns the set of tiles covering
    // a given lon/lat range.
    {this}([lon], [lat], d=day, y=year,
           p=product, value)
      := {$ if(value)
              return tm.getTiles(lon.max,
                                 lat.min,
                                 lon.min,
                                 lat.max,
                                 d, y, p);
            else return null; $};
  }
}
```

In this example, the signature is more complicated. `{`**`this`**`}(...)` means that the return value of the Java code is a set (specified by `{...}`) of Tiles (specified by **`this`**). The first two arguments, lon and lat, are surrounded by `[...]`, indicating that the variable domains should be intervals. The next three arguments, d, y, and p are defined as being equal to the Tile attributes day, year and product, not shown in this example. Finally, **`value`** is the boolean value of the `covers` relation, true if and only if the tile covers the specified lon/lat.

The Java code is also more complex. Unlike the previous example, it has a conditional and an explicit return call. If **`value`** is true, then it returns the result of the method getTiles. Since lon and lat are intervals, we refer to their maximum and minimum values to specify the bounding box of interest. If **`value`** is false, it returns **`null`**, meaning the set of tiles could not be determined, since there is no method for returning the tiles outside of a bounding box.

4.3 Function constraints

Functions, like attributes, can have constraints associated with them, the only difference being that the constraints cannot reference the keyword **`this`**, because there is no object to reference. Infix mathematical operators are also functions, and they can be referenced using a syntax similar to that used for C++ operator overloading. For example to specify that the "+" operator can be used to concatenate strings, we can write

```
static string operator+ (string s1,
                         string s2) {
  constraint Concat(value, s1, s2);
}
```

where Concat is a constraint from the constraint library, specifying that the first argument is the concatenation of the remaining arguments.

```
CTRSPEC  ::= constraint ( <IDENTIFIER>
             ( ARGS ) _;_ ) | { ( JAVACTR )+ } )
JAVACTR  ::= ( CTRARG CTRARGS _:=_
             <INLINE_CODE> _;_ )
             | ( [ CTRARG ] CTRARGS _:=_
             [ <INLINE_CODE> _,_
             <INLINE_CODE> ] ; )
             | ( { CTRARG } CTRARGS _:=_
             { <INLINE_CODE> } _;_ )
CTRARG   ::= <IDENTIFIER> | value | this
CTRARG2  ::= (CTRARG | [ CTRARG ] | { CTRARG })
             ( _=_ ADDITIVE )?
CTRARGS  ::= ( CTRARG2 [ _,_ CTRARGS ] )
```

5 Goals

Goals are used primarily to describe data products that the system should produce. Data product descriptions should specify the following four attributes:

- Data semantics: the information represented by the data. That is, what facts about the world can be inferred from the data contents.
- Data syntax: how the information is coded in the data. For example, what pixel values in an image are used to represent the information.
- Time: what time the information pertains to. For example, we need to be able to distinguish between rainfall last week and rainfall last year.
- Location: where the data file should be put or delivered.

Time is an optional argument of all fluents, and specifying location requires no special syntax. The mapping between semantics and syntax is specified using the keyword **when**. For example, to specify that a file represents the temperature over a particular region, using the LAZEA projection and the function `tempEncoding` to map from temperatures to pixel values, we could write:

```
forall int x, int y, real lon, real lat,
       real t;
when(tempEncoding(temperature(lon, lat)) = t
     && proj(LAZEA, x, y, lon, lat)
     && 0 <= x < MAXX && 0 <= y < MAXY) {
  file.pixelValue(x, y) = t;
}
```

We will call the expression inside the parentheses following the keyword **when** the left-hand side (LHS) of the goal, and we will call the expression in the braces the right-hand side (RHS).

A **when** condition describes an implication, but an implication between conditions that hold at two different times. The LHS implicitly refers to the initial state (unless an earlier time is specified), and the RHS refers to the final state (whenever the goal achieved). Because the agent cannot change the past, the only way to achieve the goal is to make sure the RHS is satisfied, subject to the conditions given by the LHS. Formally, the only difference between conditions in the LHS and conditions in the RHS is the time that they refer to. However, we follow the following conventions when using goals to describe requested data

- The semantics of the desired data (e.g., temperature) is specified in the LHS of the goal, because it concerns properties of the world that hold when the goal is specified (or earlier), properties that should not (and usually cannot) be affected by the agent.
- The data syntax (e.g., pixelValue) is specified in the RHS, because it concerns properties of data that may not exist at the time the goal is given, properties that must be affected by the agent to produce the requested data.
- Constraints (e.g., `0 <= x < MAXX`) are specified in the LHS of the goal because, being static, they must hold in the initial state and cannot be affected by the agent.
- Predicates describing semantics, syntax and constraints are all disjoint. Predicates describing syntax are static properties of static objects, which can be created but not modified.

These conventions are not explicitly enforced by the language. There are no keywords to indicate "syntactic" and "semantic" predicates, and it is entirely valid to specify a goal that relates the syntax of one file to the syntax of another. However, to ensure sensible behavior from the planner, these and similar conventions discussed below should be followed where appropriate.

```
GOAL     ::= goal <IDENTIFIER> ( PARAMS? ) {
             ( (output | forall | exists) PARAMS
             _;_ )* OREXP }
OREXP    ::= CONDEXP+ ( _||_ CONDEXP+ )*
CONDEXP  ::= ( when ( ANDEXP ) { CONDEXP* }
             ( else { CONDEXP* } )? ) | EQUAL _;_
ANDEXP   ::= EQUAL ( _&&_ ( EQUAL ) )*
EQUAL    ::= RELATION ( ( _=_ | _!=_ ) RELATION )*
RELATION ::= ADDITIVE ( ( _<_ | _>_ | _<=_ | _>=_
             ) ADDITIVE )*
ADDITIVE ::= MULTIPL ( ( _+_ | _-_ ) MULTIPL )*
MULTIPL  ::= UNARY ( ( _*_ | _/_ | _%_ ) UNARY )*
UNARY    ::= ( _+_ | _-_ | _!_ )? PRIMEXP
PRIMEXP  ::= ( ANDEXP ) | ( FUNEXP | this )
             ( _._ FUNEXP )* | LITERAL
FUNEXP   ::= <IDENTIFIER> ( ( ARGS ) )?
LITERAL  ::= <INTEGER_LITERAL>
             | <FLOATING_POINT_LITERAL>
             | <CHARACTER_LITERAL>
             | <STRING_LITERAL> | null
             | true | false
ARGS     ::= ADDITIVE (_,_ ARGS )?
```

6 Actions

Actions can include sensors (which output data based on the state of the world) and filters (which output data based on their inputs), so preconditions and effects describe inputs and outputs as well as the state of the world. Additionally,

actions must be executable, so the procedure for executing an action (i.e., Java code) is part of the action description.

```
ACTION  ::= action <IDENTIFIER> ( PARAMS ) {
            (( input | forall) PARAMS _; )
            | (output OUTPUTS _; )
            | PRECOND | EFFECT | EXEC )* }
OUTPUTS ::= PARAMDEF (copyof <IDENTIFIER>)?
            (_,_ OUTPUTS)?
```

6.1 Inputs, outputs and parameters

As in PDDL (McDermott 2000), actions are parameterized, and parameters are typed. In addition to ordinary parameters, two kinds of variables are recognized as unique and are treated somewhat differently, namely inputs and outputs.

Outputs represent objects (e.g., files) generated as a result of executing the action. An output does not exist before the corresponding action is executed, and is always distinct from all other objects.

Inputs represent objects that are required by the action but are not required to exist after the action has been executed. Inputs may come from outputs of other actions or they may be preexisting objects. In the former case, all preconditions describing attributes of a given static input must be supported by the same action, since only one action can have produced the output, and once it is created, no action can change it.

Ordinary parameters are essentially like the parameters passed to method or function calls in C or Java; they refer to primitive values or objects that may exist before the action is executed and may persist afterward. If they represent fluents, the action may change their values.

In addition to parameters, inputs and outputs, actions can refer to universally quantified variables and introduce variables corresponding to new objects with the **new** keyword, discussed in Section 6.3.

6.2 Preconditions

Preconditions describe the conditions that must be true of the world and of the inputs in order for the action to be executable. Thus, action preconditions need to reference the input variables and the prior world state, but cannot reference the output variables, which describe objects that don't exist in the prior state.

Low-level actions, such as filters, can be described purely in terms of the syntactic properties of the input files. For example, an image-processing operation doesn't care whether the pixels of the input image represent temperatures in Montana or a bowl of fruit. All that matters are the values of the pixels. Thus, the preconditions for these actions should refer only to properties of the data that hold in the prior state. Similarly, simple sensors depend only on the immediate state of the world, so their preconditions should only refer to conditions of the world that hold in the prior state.

However, some high-level actions, such as ecosystem models, expect their inputs to represent certain information about past states of the world, such as temperature or precipitation, so it is appropriate for the preconditions of these actions to specify the information content of their inputs, not just the structure, and to reference states other than the prior state. In other words, preconditions, like goals, can include metadata descriptions, which are described using the keyword **when**.

The LHS of a **when** precondition, like the LHS of a goal, refers to the initial state. The RHS, however, rather than referring to the final state, refers to the start of execution of the corresponding action. Conventions for describing data inputs in preconditions are the same as the conventions for describing goals: The LHS specifies the semantics of the data file and the RHS specifies the syntax. Any constraints should appear in the LHS.

Preconditions are introduced with the keyword **precond**, and introduce a condition, which may be disjunctive.

```
PRECOND ::= precond OREXP
```

6.3 Effects

Effects, introduced with the keyword **effect**, are used primarily to describe the outputs generated by an action. Outputs depend on the state of the world (in the case of sensory actions) or the inputs (in the case of filters), so effects need to be able to reference both the prior state and next state and both the input and output variables.

```
EFFECT ::= effect ( WHENEXP )+
```

Conditional effects Like goals and preconditions, conditional effects are introduced using the keyword **when**, but here the LHS refers to the prior state (and input variables), not the initial state. The RHS describes the next state and output variables, so the combination of the two describes how the output depends on the input (or on the state of the world).

As with goals, there are conventions for describing effects.

- Sensors are described using conditional effects in which conditions on the LHS are either constraints or fluents describing the semantics of the data, and conditions on the RHS are statics describing the syntax of the output data.
- Filters are described using conditional effects in which conditions on the LHS are either constraints or statics describing the syntax of the input data, and conditions on the RHS are statics describing the syntax of the output data.

These conventions are not enforced by the language, and it is perfectly acceptable to write other kinds of actions, including, for example, ones that change the world based on the content of an input file.

Stripping away the syntactic sugar, every atomic RHS expression involves setting the (possibly boolean) value of a function or attribute or creating a new object. A static attribute can only be set if it is an attribute of a newly created object. We depart from the C++/Java syntax and use the notation ":=" to denote assignment and "=" to denote equality.

For example, to describe a threshold action, which sets output pixels to either BLACK or WHITE, depending on whether the corresponding input pixels are below or above a given threshold `thresh`, we can write:

```
action threshold (unsigned thresh) {
  input Image in;
  output Image out copyof in;
  forall Unsigned x, Unsigned y;
  effect when ((x < in.xSize)
               && (y < in.ySize) {
    when (in.valueAt(x, y) <= thresh) {
        out.valueAt(x, y) := BLACK;
    } else {
        out.valueAt(x, y) := WHITE;
    }
  }
}
```

The keyword **else** has the same meaning as in C or Java. The keyword **copyof** is explained below.

WHENEXP	::=	(when { ANDEXP } { (WHENEXP)* } (else { (WHENEXP)* })?) | CONSEQNT
CONSEQNT	::=	ASSIGNMNT | NEWDECL
ASSIGNMNT	::=	CFUN (. CFUN)* (:= (EQUAL | NEWEXP))? ;
CFUN	::=	<IDENTIFIER> ({ CARGS })?
CARGS	::=	(ADDITIVE | NEWEXP) (, CARGS)?

Object creation and copying Output variables implicitly describe newly created objects, but it is sometimes necessary to explicitly refer to object creation in action effects. For example, an output may be a complex object, such as a file archive or a list, with an unbounded number of complex sub-elements. Since each of those sub-elements is (possibly) newly created, we need some way of describing their creation. We do so using the keyword **new**.

Additionally, newly created objects may be copies of other objects, possibly with minor changes. Listing all the ways the new objects are the same as the preexisting objects can be cumbersome and error-prone, so we would like to simply indicate that one is a copy of the other, and then specify only the ways in which they differ. We do so using the **copyof** keyword.

Suppose we have an action whose input, `in`, is a collection of JPEG files and whose output, `out`, is a new collection, in which the files from the input are compressed with quality of 0.75.

```
forall Image orig;
when(in.contains(orig)) {
  out.contains
     (new Image copyof orig {
       quality := min(orig.quality, 0.75); });
}
```

When an object is copied, all attributes of the original object are inherited by the copy, unless explicitly overridden. For example, the new Image is identical to the original in every way, except in quality, which is set to 0.75. Note that this is one way in which attributes of objects are different from other relations on objects. `in.contains(orig)` is an attribute of `in`, but not an attribute of `orig`, so after `orig` is copied, `in.contains(copy)` is not true but, for example, `copy.format = JPEG` is true.[1]

The copy and the original need not be the same type, as long as they inherit from or implement a common parent type. All attributes common to both types are copied.

NEWDECL	::=	new QUALTYPE <IDENTIFIER> (copyof <IDENTIFIER>)? (({ (ATTRIBUTES)* }) | ;)
NEWEXP	::=	new QUALTYPE (copyof <IDENTIFIER>)? (({ ATTRIBUTES * }) | ;)
ATTRIBUTES	::=	FUNEXP := (EQUAL ; | NEWEXP)

6.4 Execution

The action descriptions include instructions for actually executing the action. These instructions are written in Java, which enables us to write actions that correspond to any operation that can be performed by the Java runtime environment, including invoking methods on objects or invoking system calls. All parameters, inputs and outputs may be referenced in the Java code, though outputs, being uninitialized prior to execution, must be initialized in the Java code before being referenced.

EXEC	::=	exec <INLINE_CODE> ;

7 States

A typical component of planning problems is a specification of the "initial state," from which the goal must be achieved. In IMAGEbot, the state is stored in a database and is updated based on observations during plan execution. A significant amount of state information is also communicated through the execution of inlined code during constraint reasoning. Since many aspects of the world are highly dynamic, we rely more on information obtained dynamically as a result of sensing than on static information contained in a file.

However, there is still value in being able to provide static state information, especially concerning metadata descriptions for stable data sources. The language provides the ability to define multiple named states through the **state** keyword. States may be thought of as dumbed-down actions that have no preconditions and can only be "executed" in the initial state. As with goals, metadata descriptions are specified using the **when** keyword. As with goals, the LHS can refer to the initial state or earlier, but the RHS refers to the initial state, not the final state. The conventions for describing the semantics and syntax of data are the same as they are for goal descriptions. State information stored in the database, which includes information the agent has "learned" over time, has the same form as that introduced using the **state** keyword.

STATE	::=	state <IDENTIFIER> { (forall PARAMS ;) | WHENEXP)+ }

[1] In contrast, in ADLIM, all relations involving the original were assumed to hold for the copy unless explicitly overridden, so it would be necessary to declare that `in.contains(copy)` is false after copying `orig` to `copy`.

8 Conclusions and Related Work

We have described DPADL, an action language for data processing domains, which is used in the IMAGEbot system. The parser for the language, and a planner that supports the language, are fully implemented, and we are developing DPADL descriptions for the TOPS ecosystem forecasting system. Currently, a subset of TOPS, dealing with MODIS satellite data, is supported.

The focus of this paper has been to describe DPADL as a programming language, focusing on language syntax and features rather than semantics and formal properties. This is based on our belief that writing planning domains to automate data processing tasks is at its heart programming. Our planner-based approach reduces the need for programming from writing one script for each task to writing one domain description for all tasks,[2] but it is still necessary to "program" the one domain. The language should support the features needed to program effectively.

Throughout the paper, we have described conventions for describing data and data-processing actions. These conventions are not enforced at the language level, because doing so would limit the flexibility of the language without giving much advantage, but we have found that following these rules makes it simple for a planner to infer the effects of chaining multiple data-processing actions together. "Simpler" representations, in which data semantics is encoded directly in (the RHS of) a goal or effect, conversely, make it much less obvious what should happen when actions are chained together.

It is always attractive to use a standard language rather than defining a new one, and there are popular languages for describing planning domains, web services and metadata that we considered.

PDDL (McDermott 2000), the language devised for the AIPS programming competitions, has become a standard for describing planning domains. Unfortunately, PDDL is not ideal for describing data processing domains. It provides no support for object creation or copying, explicit inputs or outputs, metadata, or integration with a run-time environment or constraint reasoning system. There are also syntactic disadvantages, such as lack of object-oriented notation, which could be worked around but are obstacles to clean domain descriptions. We opted instead to base our language on the well-known syntax of C++ and Java, since many programmers are familiar with that syntax, reducing the learning curve for the language. An additional motivation was that action descriptions and program code both describe the same things — state change, conditional on the current state, so using similar syntax for both is appealing.

DAML-S (Ankolenkar *et al.* 2002) and WSDL (Christensen *et al.* 2002) are languages for describing web services, both based on XML. DAML-S is the more expressive, allowing the specification of types using a description logic and allowing one to specify preconditions and postconditions, which might be used by a planning agent. However, we don't believe that description logics are expressive enough to describe the data-processing operations that we need to support. Another disadvantage of these languages from a usability perspective is that XML files tend to be verbose and difficult for humans to read. XML is ideal for storing information in a form that can readily be shared by computer programs, but it is human-readable only in the limited sense that it is written in ASCII text.

The Earth Science Markup Language (ESML; http://esml.itsc.uah.edu) is another language based on XML, under development at the University of Alabama in Huntsville to provide metadata descriptions for Earth Science data. Unlike DAML-S and WSDL, ESML is well suited to describing the complex data structures that appear in scientific data. Unlike DPADL, it is only intended to describe data files, not data processing operations, but it does provide explicit support for describing the syntax and semantics of data files and allows the specification of constraints in the form of equations. Although it is less expressive and more specialized than DPADL, it is a promising metadata standard for Earth Science. In the near future, we hope to support conversion between ESML and DPADL metadata specifications.

Near the far end of the expressiveness spectrum, the situation calculus (McCarthy & Hayes 1969) provides plenty of expressive power, but at a price: planning requires first-order theorem proving. We opted instead to make our language as simple as possible, but no more so. DPADL does not support domain axioms, nondeterministic effects or uncertainty expressed in terms of possible worlds, and much of the apparent complexity of the language is handled by a compiler, which reduces complex expressions into primitives that the planner can cope with. Despite the superficial similarity to program synthesis (Stickel *et al.* 1994), DPADL action descriptions are not expressive enough to describe arbitrary program elements, and the plans themselves do not contain loops or conditionals.

Of the many planning domain description languages that have been devised, the closest to DPADL is ADLIM (Golden 2000), on which it is based. Advances over ADLIM include tight integration with the run-time environment (Java) and constraint system and a Java-like object-oriented syntax that makes it natural to describe objects and their properties. As discussed in Sections 2 and 6.3, this is not just syntactic sugar, but encodes valuable information used by the planner.

Collage (Lansky & Philpot 1993) and MVP (Chien *et al.* 1997) were planners that automated image manipulation tasks. However, they didn't focus as much on accurate causal models of data processing, so their representation requirements were simpler.

The EnVironmEnt for On-Board Processing (EVE) (Tanner *et al.* 2001) is an execution framework for data-processing plans to be run on-board an Earth-orbiting satellite. Unlike IMAGEbot, EVE provides no planning capabilities; plans are generated by humans.

Acknowledgments

I am indebted to Wanlin Pang, Jeremy Frank, Ellen Spertus and Petr Votava for helpful comments and discussions. This work was funded by the NASA Computing, Information and Communication Technologies (CICT) Intelligent

[2] In the words of Dan Weld, "A softbot is worth a thousand shell scripts."

Systems program.

References

Ankolenkar, A.; Burnstein, M.; Hobbs, J. R.; Lassila, O.; Martin, D. L.; McDermott, D.; McIlraith, S. A.; Narayana, S.; Paolucci, M.; Payne, T. R.; and Sycara, K. 2002. DAML-S: Web service description for the semantic web. In *Proceedings of the 1st Int'l Semantic Web Conference (ISWC)*.

Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*.

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. 2002. Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium. Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

Golden, K., and Frank, J. 2002. Universal quantification in a constraint-based planner. In *Proc. 6th Intl. Conf. AI Planning Systems*.

Golden, K. 2000. Acting on information: a plan language for manipulating data. In *Proceedings of the 2nd NASA Intl. Planning and Scheduling workshop*, 28–33. Published as NASA Conference Proceedings NASA/CP-2000-209590.

Lansky, A. L., and Philpot, A. G. 1993. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*.

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*. Edinburgh University Press. 463–502.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 2(2):35–55.

Nemani, R.; Votava, P.; Roads, J.; White, M.; Thornton, P.; and Coughlan, J. 2002. Terrestrial observation and predition system: Integration of satellite and surface weather observations with ecosystem models. In *Proceedings of the 2002 International Geoscience and Remote Sensing Symposium (IGARSS)*.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction*.

Tanner, S.; Keiser, K.; Conover, H.; Hardin, D.; Graves, S.; Regner, K.; Wohlman, R.; Ramachandran, R.; and Smith, M. 2001. EVE: An on-orbit data mining testbed. In *Proceedings of the IJCAI workshop on Knowledge Discovery from Distributed, Heterogeneous, Autonomous, Dynamic Data and Knowledge Sources*.